Fast Analytical Motion Blur with Transparency

Citation for the original published paper (version of record):
Rönnow, M., Assarsson, U., Fratarcangeli, M. (2021). Fast Analytical Motion Blur with Transparency. Computers and Graphics, 95: 36-34. http://dx.doi.org/10.1016/j.cag.2021.01.006

Computers & Graphics 95 (2021) 36–46

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Fast analytical motion blur with transparency ☆

Mads J.L. Rønnow*, Ulf Assarsson, Marco Fratarcangeli

Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 16 September 2020
Revised 26 November 2020
Accepted 15 January 2021
Available online 23 January 2021

Keywords:
Real-time rendering
Motion blur
Parallel computing

ABSTRACT

We introduce a practical parallel technique to achieve real-time motion blur for textured and semi-transparent triangles with high accuracy using modern commodity GPUs. In our approach, moving triangles are represented as prisms. Each prism is bounded by the initial and final position of the triangle during one animation frame and three bilinear patches on the sides. Each prism covers a number of pixels for a certain amount of time according to its trajectory on the screen. We efficiently find, store and sort the list of prisms covering each pixel including the amount of time the pixel is covered by each prism. This information, together with the color, texture, normal, and transparency of the pixel, is used to resolve its final color. We demonstrate the performance, scalability, and generality of our approach in a number of test scenarios, showing that it achieves a visual quality practically indistinguishable from the ground truth in a matter of just a few milliseconds, including rendering of textured and transparent objects. A supplementary video has been made available online.[1]

1. Introduction

The Visual Effect industry (VFX) is currently undergoing a paradigm shift towards real-time content productions. Modern game engines, in fact, are able to reproduce many realistic graphics techniques interactively which traditionally took a high amount of computation time. As a consequence, design and prototyping cycles are shorter, and more content can be produced in less time saving costly resources. Being able to produce vast amounts of content quickly is also useful to create training datasets for neural networks from scratch.

To reach interactivity, however, some of these effects are still grossly approximated leading to visible artifacts. One of these effects is *motion blur*, which is essential to represent moving objects. Motion blur is a common optical effect in photographs and videos that occurs when the positions of objects change with respect to the camera point of view during the interval in time where the camera shutter is open. If the objects are moving rapidly, or the shutter interval is long enough, then the objects leave a blurred streak in the direction of motion. It is important to reproduce this effect to synthesize immersive and more believable scenes, mimic specific camera models, or achieve artistic effects.

The computation time for this type of effect is particularly critical for real-time interactive graphics, such as video games, where the time budget available for rendering effects such as motion blur is just a few milliseconds. For this reason, the approach used in modern game engines is to use computationally-cheap, screen-space approaches in post-processing to achieve motion blur (e.g., [1]). While being fast, these methods suffer from occlusion issues and artifacts, in particular when used with transparent geometries, or when background and foreground objects move in conflicting directions.

Gribel et al. [2] provided precise directions on how to accurately represent motion blur in a computer animation, which is generic enough to represent any type of triangulated object, including transparent, textured, shaded and any of these combined. The method is validated only in a software renderer though, and it is not possible to directly map it to modern graphics hardware to use it in a real-time application.

In this work, we start from such directions and provide a practical, efficient parallel implementation for a corresponding GPU-based algorithm. Instead of using analytic edge equations and depth functions for moving triangles [2], we represent each triangle trajectory, during each single time step, by a prism with uv coordinates and t values stored for each prism vertex. Conservative 2D hulls of these prisms are then rasterized using a shader that computes the entry and exit points of the prism for a ray from the camera through the pixel. These points are pairwise connected into intervals inside the prism, for which time, depth, and also texture coordinates (extracted from the intersection points on the prism

☆ This paper was recommended for publication by Michael Doggett.

* Corresponding author.
E-mail address: ronnow@chalmers.se (M.J.L. Rønnow).

[1] Supplementary video available here

https://doi.org/10.1016/j.cag.2021.01.006
0097-8493/

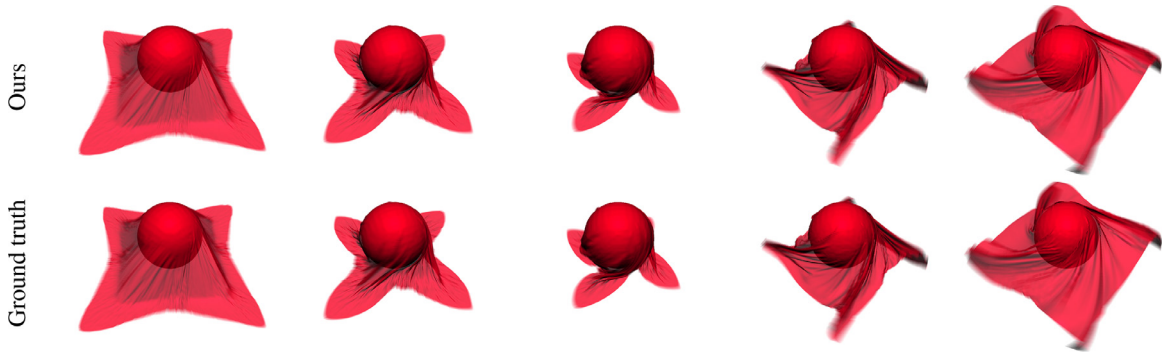

Fig. 1. A transparent cloth falls on top of a rotating sphere producing motion blurred images. Our results (*top*) are rendered in real-time and are visually indistinguishable from the ground truth (*bottom*).

surface) are stored for both the start and end points of the interval. The intervals are then quickly sorted on time for each pixel, followed by a pass that resolves the final colors by sweeping the list of time-sorted intervals while using an active list of intervals dynamically sorted on depth [2]. We extend this step to handle textures during the color and transparency aggregation and present an efficient CUDA implementation. The main benefits of our approach are:

- *Performance*: our method fully exploits the massively parallel capabilities of modern GPUs, achieving a performance suitable for interactive graphics;
- *Scalability*: we instantiate a thread for each pixel covered by a moving object. Since each pixel is handled by a single thread on the GPU, this makes our approach scalable in the number of threads that the graphics card can instantiate;
- *Generality*: differently from previous GPU-based works [3], our approach can handle triangulated textured objects that are also transparent.

2. Related work

In this section, we first provide a brief overview of the related techniques in motion blur and conclude by motivating our own approach.

The first brute-force approaches proposed in the literature blurred the current animation frame with previous ones [4,5]. These approaches are simple to implement and lead to accurate results. The number of required previous frames, however, may become quite large leading to a loss of performance in particular for high pixel resolutions.

Stochastic sampling approaches (e.g., [2,6–11]) exploit computational features hardwired on modern GPUs to randomly sample the triangles occluding a pixel both in time and space. Stochastic methods are more efficient than the brute force approach, but still require many samples per frame and tend to suffer from sampling noise, which is magnified as the per-frame length of the motion increases. Accuracy is achieved by increasing the number of samples but this, in general, negatively affects their performance, making them suitable only for offline productions like movies.

Post-processing approaches (e.g., [1,12,13]) are fast and suitable for hard time budget applications and, for this reason, are widely employed in modern game engines. In these approaches, the dominant velocity of the triangles occluding a pixel is included in the attributes of the pixel itself, and used to blur in screen space. While these highly-parallel approaches are scalable and fast, they may suffer from the lack of robustness typical of screen-space approaches due to the loss of information caused by the projection and rasterization from the 3D scene to the 2D image space. This may lead to artifacts such as incorrect blurring of the background, or errors when different moving objects traverse the same pixel in different directions.

A promising avenue for 3D post-processing approaches are learning-based techniques [14,15]. High performance is achieved by using different flavors of neural networks to convert pixel attributes (e.g., position, normal and color) to a number of screen-space effects, e.g., ambient occlusion, light scattering and motion blur. Recently, motion blur effects have been also applied to still images as an artistic style to convey motion and to direct attention [16–18]. Recent developments in GPU hardware is enabling real-time ray tracing, including support for ray-traced motion blur [19]. Though these hardware features still use stochastic sampling that needs high sampling counts to achieve the same image quality as analytical methods. Hybrid approaches, however, might be a viable path forward for analytical motion blur.

In this work, we provide a practical, GPU-based implementation of the theory provided by Gribel et al., which described a method for analytically rendering motion blur using triangle-edge equations with a time parameter [2]. In the same work, they also proposed a compression scheme removing the need for sorting unbounded per-pixel lists, together with a method for accurately blending different layers of colored triangles. The method was only validated on a software renderer and with a single color per triangle. Extensions to this work were made still using software rasterization [20]. The same blending technique was used to implement a GPU version for opaque geometry only, not supporting transparent objects [3]. The actual performance of this method is unclear, mostly because it is not measured with respect to the main bottlenecks, namely the depth complexity of each pixel and how much each triangle moves on the screen for each frame.

Our system optimizes for such factors, obtaining real-time performances while maintaining an accuracy close to ground truth. In fact, we use uncompressed time values (limited to 16-bit floating point numbers), and represents the potentially curved edges of the moving triangle accurately with bilinear patches. Our approach scales well with the number of available threads. We test our approach on a number of challenging scenes, achieving real-time performances even for high polygonal and pixel resolutions.

3. Overview

Let us assume our animation to be continuous in time instead of being a mere sequence of frames separated by discrete time steps. Let us also assume that the considered time span is defined between the instants when the camera shutter opens and closes, normalized between $t = 0$ and $t = 1$. Since time is continuous in the considered time step, we would obtain an infinite sequence of rasterized images. In this ideal setting, the motion blurred image in a given time is simply the result of averaging together the

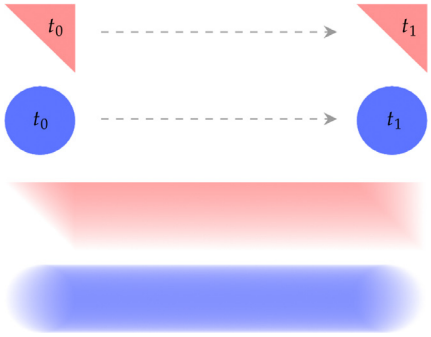

Fig. 2. Two objects moving across the screen from left to right between two frames, without motion blur (top) and with motion blur (bottom). The pixels at the start and end positions are less saturated than the ones towards the center, because the objects cover those pixels for a shorter amount of time.

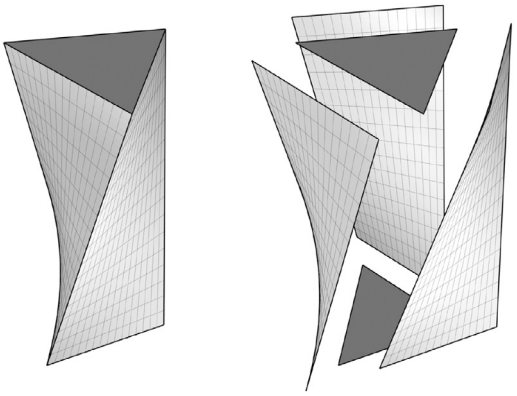

Fig. 3. *Left.* A triangular prism shape represents the trajectory of a moving triangle. *Right.* The prism has been cut open to reveal the two triangles and three bilinear patches it is composed of in our representation. The dark grey triangles represent the start and end positions of the triangle's motion.

infinite number of images. It is important to note that the color of a pixel is given by the color of the triangles covering it, weighted by the quantity of time the triangle covers the pixel while moving. A fast-moving triangle traversing the screen, for example, will have a small influence on the color of the covered pixels. A static, opaque triangle in front of the scene will define entirely the final color of the pixel.

In a computer animation, motion is discrete, simulated by time steps, rather than being continuous. In this case, motion blurred images can be computed by averaging together a finite set of images produced with small increments in time between the opening and closing time of the shutter. This brute force approach scales poorly with the image resolution and it can be highly inefficient [4,5]. Nonetheless, the quality of the motion blur is high and we consider it as our ground truth. If, for example, we consider N images while the shutter is open, and if a triangle is only present in a pixel for a single step, then that pixel's color will be 1/N of the triangle's color. This effect is shown in Fig. 2.

In a scene with many triangles, they may occlude each other when they move. With standard transparent rasterization, if an occluder is opaque, then the occludee will be invisible. If the occluder is semi-transparent, the occludee will be partially visible, i.e., the depth order of triangle fragments determines the final pixel color. For motion blur, occlusion is handled similarly by the ground truth method described above. Depth testing can be used for opaque geometry, and blending with either the OVER or the UNDER operator can be used for transparent geometry [21]. The depth order of triangles can change during the shutter window, and a triangle may be occluded in some of the incremental steps but not in others.

Similar to previous work [3,22,23], we represent the triangle trajectory with a prism, as depicted in Fig. 3 (left). The prism is constructed from the triangle's start and end position together with bilinear patches from the extrusion of the triangle edges along the linear vertex motions.

We cast a ray from a pixel center along the z-axis and store the surface properties at the entry and exit intersection points of the prism. Interpolated time values are embedded on the prism surface as one of the surface properties, which we use to find the time span for the triangle's presence in the pixel. Other properties include clip-space depth, UVs, normal, and texture ID. Each of these (apart from the texture ID), as well as the time property, are linearly interpolated between their respective values of the start and end triangles. Linear interpolation is not fully physically accurate, as discussed in Section 5.1.

Our system implements analytical motion blur efficiently on the GPU and is composed of four steps (Fig. 4):

Step 1. Bound the screen-space area of a moving triangle In order to bound, for each prism, the pixels that should cast prism intersecting rays, we compute a conservative clip-space hull of the prism. In a single parallel step, we compute the clip-space 2D AABB of every moving triangle.

Step 2. Render moving triangles The AABB of each moving triangle is rasterized with a fragment shader performing ray casting to find each primary ray's entry and exit points of the prism. The shader also pairs the intersection points, based on their time order, into intervals and outputs them to a buffer capable of storing an array per pixel. This involves a pre-pass. First, the depth complexities for all pixels is established. An exclusive sum over the depth complexities is computed to determine the start location of each pixel array as well as the buffer size needed to fit exactly all of them.

Step 3. Sort intervals by time The intervals for each pixel are sorted by entry time in order to find triangles that overlap in time within a pixel.

Step 4. Pixel color resolve This pass resolves the color contribution of each interval based on time overlap and occlusion with other intervals, and the duration of time a triangle is present within a pixel. This is combined with anisotropic texture lookups to get the linearly interpolated texture colors based on the entry and exit UVs of the triangles.

The theory of our method regarding rendering of motion blurred geometry is largely based on the work by Gribel et al. [2]. A short description of the method will be presented in the following section. We omit compression and aim for a more precise solution. An efficient GPU implementation is described, with support for transparency and texturing.

4. Method

4.1. Rendering prisms

We represent the surface of the prism formed by a moving triangle directly with a set of triangles and bilinear patches as depicted on Fig. 3 (right). In previous works, the prism sides are approximated with triangles [3,22]. Our method, however, models the prism sides accurately with bilinear patches.

In order to render a bilinear patch, we need to find a convex set of vertices that conservatively contain its clip-space surface. In practice, we use a single parallel step to compute an AABB of the six clip-space vertices formed by each moving triangle's start and end positions that conservatively contains all five sides of the the prism as sketched in (Fig. 4, step 1). The AABB is subsequently

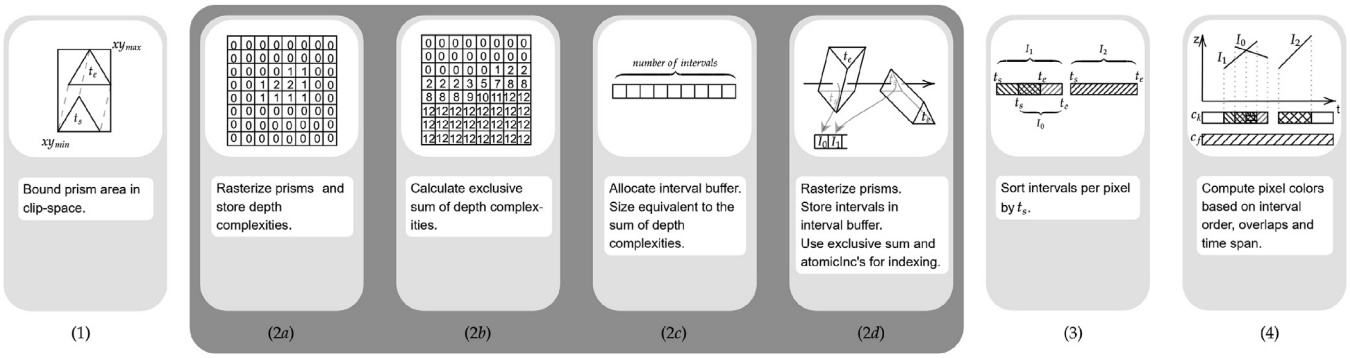

Fig. 4. The stages of our motion blur system.

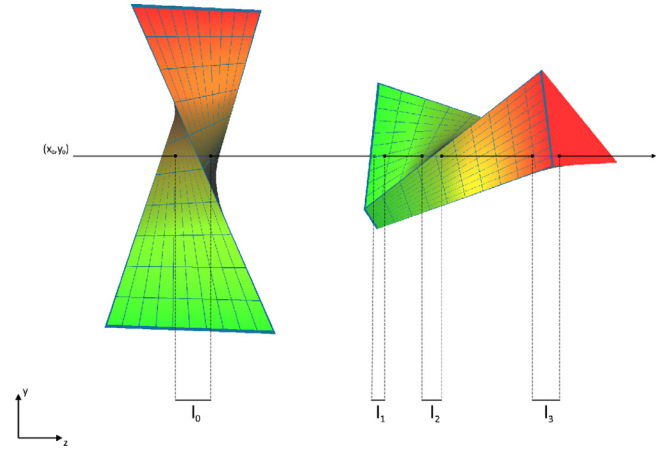

Fig. 5. As a ray intersects with a prism there can be multiple intervals. The ray intersects the left prism at two distinct points creating one interval I_0, while on the right the ray intersects the prism at six distinct points resulting in three intervals I_1, I_2, and I_3. The intersection points per-prism are sorted by time and paired up into intervals. Time values between 0 and 1 are embedded on the prism surfaces, illustrated here as a color from green to red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

t_s	t_e	z_s	z_e	uv_s	uv_e	$normal_s$	$texID$
16 *bits*	16 *bits*	32 *bits*	32 *bits*	16 *bits*	16 *bits*	24 *bits*	8 *bits*
Interval : 160 *bits*							

Fig. 6. An interval is defined by a start time (t_s), end time (t_e), start depth (z_s), end depth (z_e), start UV (uv_s), end UV (uv_e), start normal ($normal_s$), and a texture ID ($texID$).

rasterized, and for each fragment a ray is extended from the fragment center (x_0, y_0) along the z-axis potentially intersecting prism surfaces along its trajectory as shown in Fig. 5. To find the intersections between the ray and the prism, we compute two ray/triangle intersection tests and three ray/bilinear patch intersection tests.

A ray can intersect a bilinear patch twice and a triangle once, in total up to 8 intersections can occur between a ray and a prism. While Gribel et al. [2] found the intersection points by solving time-dependent edge equations in a closed form, we found that ray/triangle and ray/bilinear patch intersection computations to be more practical and straightforward to implement. We use the ray/bilinear patch intersection algorithm described by Reshetov [24] which, according to the author, achieves better relative performance compared to approximating the bilinear patch with two triangles. Since a triangle is a special case of a bilinear patch we can even use the same intersection test for all five surfaces of the prism.

We handle all five intersection tests in a single fragment shader invocation because it simplifies the following phases of our method, in particular the sorting step. For the intersection tests, we consider all the three clip-space bilinear patches together with the two clip-space triangles when computing the convex set. We also considered using convex hulls as was done by McGuire et al. [25], but the AABB is robust and cheap enough in our experience, and the amount of exceeding pixels leads to a negligible performance penalty.

The prisms are rendered with two render passes similar to order-independent transparency approaches [26] (Fig. 4, step 2). In both render passes, the AABBs are rasterized as two triangles forming a planar quad. For each AABB, the six clip-space vertex positions, the vertex normals, and the three vertex UV coordinates of the moving triangle are passed along in the shader pipeline without interpolation.

These values are used in the fragment shader to compute three ray/bilinear patch intersection tests and two ray/triangle intersection tests. The intersection tests output intersection samples that include the depth, time, normal, and UV coordinates of the triangle or bilinear patch at the point of intersection. In the first render pass (2*a*), used to establish per-pixel depth complexities to know the required per-pixel array lengths, these sample values are not computed and instead a counter is simply incremented whenever an intersection is found, while in the second pass (2*d*) they are stored in temporary arrays for each per-pixel fragment shader invocation, then sorted by their relative time values and paired into intervals. The ray/triangle intersection samples are either at $t = 0$ or $t = 1$, and hence do not need to be sorted by time if they are explicitly placed before and after the bilinear patch intersection samples. The ray/bilinear patch tests can have two intersection samples each, in total giving a maximum of six bilinear patch samples that need to be sorted by time.

The first render pass is followed by an exclusive sum computation over the depth complexity of every pixel (2*b*), and the allocation of required GPU memory for a global interval buffer (used to store intervals in the subsequent render pass) using the sum of the per-pixel depth complexities (2*c*). The second render pass (2*d*) stores the prism intervals in the global interval buffer which was allocated in the previous substage, with indexing based on the exclusive sum of the depth complexities as well as atomic counters that count the number of intervals stored so far for each pixel. At least two intersection sample points are produced for each pixel with a prism covering it: an entry sample and an exit sample which define the interval in time and depth where the prism is present in the pixel. Each prism can in theory have up to four such

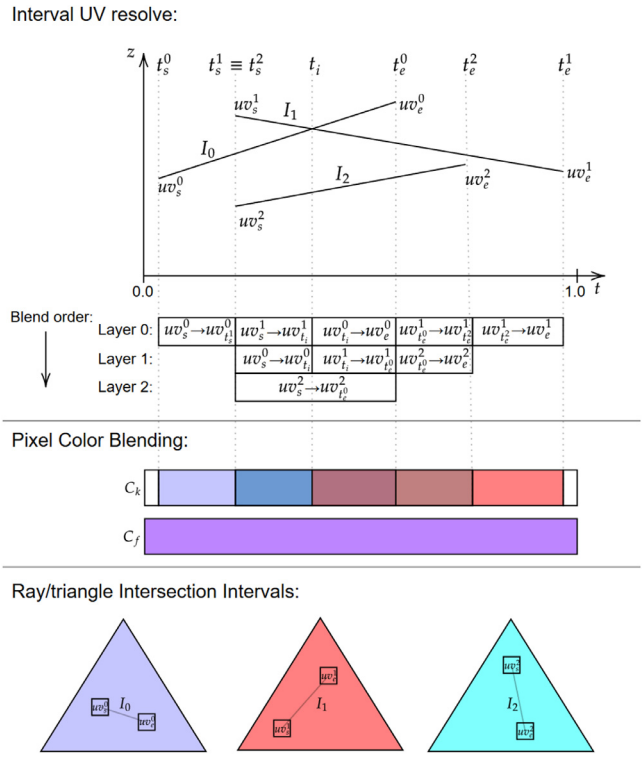

Fig. 7. Color resolve for a single pixel. At the top in *Interval UV resolve*, the intervals are partitioned based on time and depth order due to blending order requirements. In *Pixel Color Blending*, the partitioned UV ranges are used to look up the texture color for each of the time partitions. These intermediate values are shown in C_k. The final pixel color is shown in C_f as the result of averaging all the partitions together. In *Ray/triangle Intersection Intervals*, the textured triangles are shown along with the entry and exit point for the intersecting ray from which the intervals are composed. For illustrative simplicity, each interval belongs to its own triangle and each triangle has a single color, each with some degree of transparency.

intervals within a pixel. The samples are found with the ray intersection tests described earlier and as mentioned above, the intersection samples between a ray and a prism need to be sorted by time. This is in order to efficiently pair them up into intervals based on time order. An interval is 160 bits wide and is defined as shown in Fig. 6.

For ray/triangle intersections, the three vertices of the triangle and the barycentric coordinates at the hit point can be used to

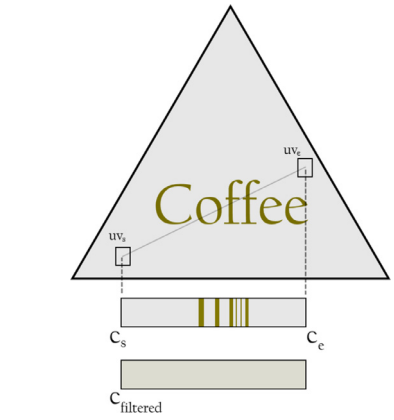

Fig. 8. A triangle with a texture depicting the text *Coffee* with a grey background. The texel colors are filtered between an interval's start UV (uv_s) and end UV (uv_e) resulting in the filtered color showed at the bottom.

interpolate the UV coordinates, normal and depth value, while the time value is either 0 or 1 for the starting position or end position triangle respectively. For ray/bilinear patch intersections, however, it is necessary to bilinearly interpolate based on the four vertices of the patch and the bilinear coordinates at the hit point. We store only the start normal in order to save memory space.

4.2. Sorting intervals

In preparation for the color resolve, the intervals for each pixel are sorted by t_s (Fig. 4, step 3). We use the work by Hou et al. [27] (modified for our use case by key-only based sorting and optimized by using CUB functions [28] for histogram and exclusive sum computation) for a segmented sort that sorts all intervals in the global interval buffer segmented by which pixel they belong to.

4.3. Color resolve

We use a method similar to the one described by Gribel et al. [2] to compute the final pixel color (Fig. 4, step 4), as detailed on Fig. 7 . We have extended the method to enable texture-mapped triangles. With texturing, intervals can no longer be assumed to have a static color from start to end, but instead the color is based on a continuous range of texels limited by the start

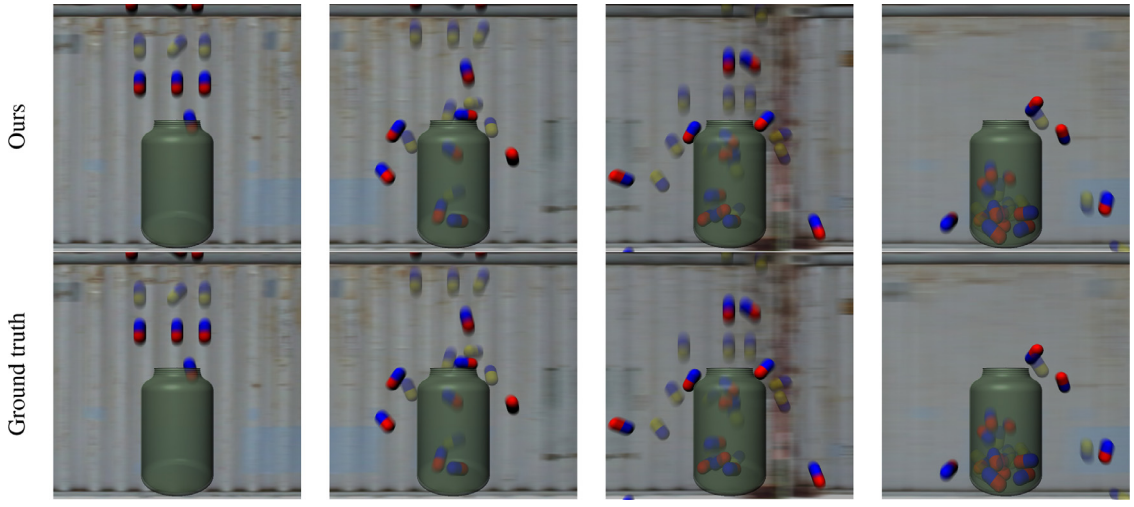

Fig. 9. Quality comparison with ground truth. Opaque and transparent pills fall into a transparent bottle while a background object moves to the left.

Algorithm 1: Per-pixel color resolve algorithm. (Optional steps for colliding triangles are marked in red).

```
Input  : Intervals for this pixel (sorted by t_s): IL,
         Active-interval list (empty at init): AL
Output: Final pixel color C_final
Initialize:
    C_final = (0,0,0);
    r_s = 0.0, r_e = 1.0 ;          // range of current time partition
    index = 0 ;                     // index to interval in IL
while index < IL.size() or AL.size() > 0 do
    if AL.size() = 0 then r_s = IL[index].t_s;// next t_s in IL
    // Find nearest resolve time r_e (the next interval start, end
    or intersect event).
    while index < IL.size() and IL[index].t_s = r_s do
        // Loop to solve IL intervals with equal t_s.r_e = min(r_e,
        IL[index].t_e);
        AL.insert(IL[index]) sorted by interval depth;
        index++;
    // Check if the next IL interval's t_s is the new nearest
    resolve time r_e:if index < IL.size() and IL[index].t_s < r_e
    then
        r_e = IL[index].t_s;
    p = FindNearestIntervalIntersection(AL);
    i_e = 0; j_e = 0;
    if p.hasIntersection and p.t > r_s and p.t < r_e then
        i_e = p.intervalALIndex_i;
        j_e = p.intervalALIndex_j;
        r_e = p.t;      // time at intersection point
    // All intervals for time range r_s - r_e are now in AL, with
    no intersections within this range.
    // Blend AL interval colors (after texture lookups and
    shading computations) front-to-back. If transmittance
    threshold is reached, stop early.
    C_k = ResolveIntervalsRange(AL, r_s, r_e, lightPos);
    C_final += (r_e - r_s) · C_k;
    // Colors have now been resolved up until time r_e.// Swap
    intersecting intervals at r_e for correct blend order in next
    partition time range:
    swap(AL[j_e], AL[i_e]); // Unlikely >1 intersection at r_e
    Remove all intervals I_i from AL where I_i.t_e ≤ r_e.
    r_s = r_e; // Advance to next time partition
    r_e = (isEmpty(AL)) ? 1.0 : min(I_i.t_e for all intervals I_i in
    AL); // accelerated by tracking current min(I_i.t_e) during AL
    interval removal above.
```

and end texture UV coordinates stored in the interval as illustrated on Fig. 8. The intervals for each pixel are sequentially resolved from $t = 0$ to $t = 1$ using an active interval list where the active intervals are kept sorted by depth.

There are two main cases that complicate the color resolve: intervals that partially overlap in time and intervals that intersect. In Fig. 7, intervals I_0 and I_1 intersect at t_i where they share depth and time values. In order to ensure the correct blending order of the two intersecting intervals, they are partitioned at the intersection point into four new intervals. To the left of the intersection point, I_1 is behind I_0, while to the right, I_0 is behind I_1. Intervals must also be partitioned when another interval starts or ends within its time range, such as I_2 ending within the time range of I_1. This is necessary in order to be able to blend the colors of I_1 and I_2 in the range where they share time, and to not blend them in the range where only one of them is present. When an interval is partitioned, its UV values are partitioned as well by interpolation.

The blending order is swapped at intersection points, as is the case in Fig. 7 between intervals I_0 and I_1 at the intersection point t_i. The boxes with $UV_a^I \rightarrow UV_b^I$ should be interpreted as: *the texture coordinates for the anisotropic texture lookup should be the UV values from UV_a^I to UV_b^I*, where I identifies the interval and a and b distinguish either the interval's start UV $\left(UV_s^I\right)$, end UV $\left(UV_e^I\right)$ or, as a result of partitioning, an interpolated UV value:

$$UV_t^I = lerp\left(UV_s^I, UV_e^I, \frac{t - t_s^I}{t_e^I - t_s^I}\right)$$

Since the color contribution of each interval depends on the UV coordinates at the start and end of the interval, partitioning an interval will change its color contribution as the UV coordinates are also partitioned. Therefore, the color contribution of each partition, including lighting computations, cannot be trivially resolved when the interval is created but must be resolved at the partitioning stage. The UV coordinates and the interpolated vertex normal need to be stored in the interval, as the color value alone is not sufficient. The partitioning of intervals and the color resolve is done with an active list approach as outlined in Algorithm 1. The CUDA source code is provided on the web, including details on further low-level optimizations. We approximate the continuous range of texels between the start and end UVs using anisotropic filtering. For increased precision, the range is split up in two $\times 16$ anisotropic lookups to simulate $\times 32$ anisotropic filtering. Similarly to Shkurko et al. [22], we assume that a ray moves linearly over a triangle surface, which makes it possible to calculate hit point data by linear interpolating two end points of an interval.

An interval intersection, such as the one depicted at the top in Fig. 7 as t_i, happens at a point where two intervals have equal time and equal depth. This occurs precisely when the triangles collide in the 3D world.

Table 1
Performance results. The values displayed are averages over all frames in the sequence, while *Max. GPU memory* is the maximum allocated GPU memory over the entire sequence. Resolutions used are: 1024×1024, 1920×1080 (1080p), 2048×2048, and 3840×2160 (4K).

	Clothball 63k tris 315k prism faces		Character running 49k tris 245k prism faces		Character dancing 49k tris 245k prism faces		Falling Pills 45k tris 225k prism faces	
	1024×1024	2048×2048	1080p	4K	1024×1024	2048×2048	1080p	4K
Time per frame (ms)	9.4	27.4	24.5	53.7	19.3	41	8.3	29.2
Max. intervals per pixel	127.8	130.9	525.1	530.8	450.5	455.4	40.5	41.1
Number of intervals $(\times 10^6)$	2.45	9.79	2.28	9.11	1.75	7.0	3.8	15.2
Max. GPU memory (MB)	175	688	189	746	133	523	281	1116

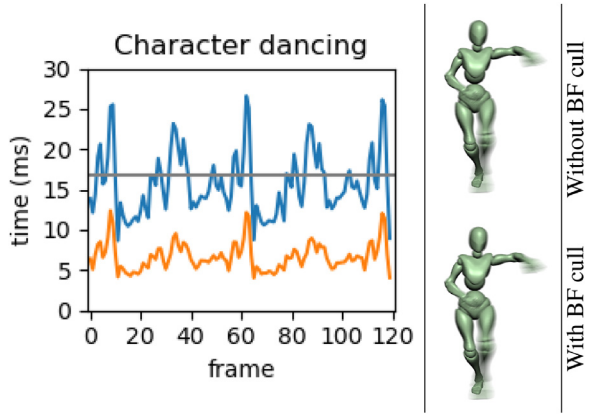

Fig. 10. Performance comparison of using back-face culling on the character dancing scene with a 1024×1024 resolution and *opaque* textures. The blue graph shows the time per frame without back-face culling and the orange graph shows the time per frame with back-face culling. As expected, the average time per frame is significantly shorter with back-face culling at 6.67 ms compared to 15.72 ms without. The grey horizontal line indicates 60 FPS. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.4. Back-face culling

For semi-transparent objects we need to store all surface samples, while for opaque objects we can discard back facing samples. While our main contribution is an efficient general system for analytical motion blur for transparent objects, mixed scenes with both transparent and opaque objects are common, which makes it important to have an implementation that can efficiently render both within a scene. For this reason we have implemented an optional, non-conservative back-face culling method that discards prisms when the triangle is back-facing at both the start and the end positions (while precise methods do exist [10]). We used back-face culling on all opaque geometry.

5. Results

We have implemented and tested our motion blur system in OpenGL 4.6 and C++/CUDA 10.2 on an NVIDIA RTX 2080 system running Windows 10. We tested four scenes with varying motion and fidelity: *clothball*,*character running*, *character dancing*, and *falling pills*. The tests have been performed at several different pixel resolutions: 1920×1080, 1024×1024, 2048×2048, and 3840×2160.

The clothball scene shows a transparent cloth falling on a rotating opaque sphere recorded at window resolutions of 1024×1024 and 2048×2048. The two character scenes both show a moving transparent character model. The character running was recorded at resolutions of 1920×1080 and 3840×2160, while the character dancing was recorded at 1024×1024 and 2048×2048. The falling pills scene shows a mix of opaque and transparent pills falling down in a transparent bottle while a background object is scrolling towards the left, recorded at 1920×1080 and 3840×2160. Back-face culling can only be used on opaque objects and is thus used only in the falling pills scene on the opaque pills and the background object, and in the clothball scene on the opaque sphere. As shown in Fig. 10, back-face culling on an opaque version of the character dancing scene significantly increases performance with no noticeable visual errors introduced by the approximate back-facing determination.

While all our scene benchmarks were run with interval intersection handling on, in most cases intersections can be ignored without a significant loss in visual quality, because triangle collisions typically would be handled by a collision detection system before rendering the scene. If intersections are ignored,

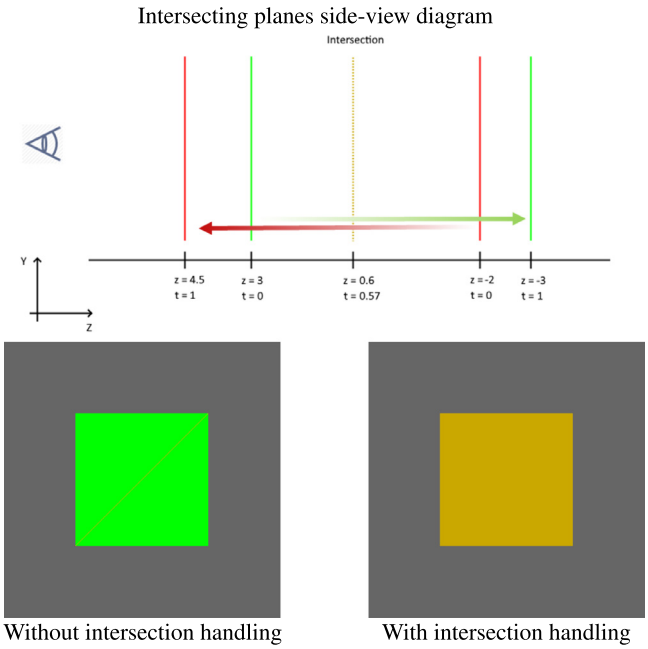

Fig. 11. *Top.* A red and a green quad are in front of an orthographic camera and move along the *z*-axis. The red one moves from back to front, while the green one moves from front to back. *Bottom.* Comparison of the results obtained with and without intersection handling. Without intersection handling (*left*), the resulting color belongs to the green quad entirely occluding the red one. With intersection handling (*right*), the colors are instead correctly blended resulting in the yellow color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

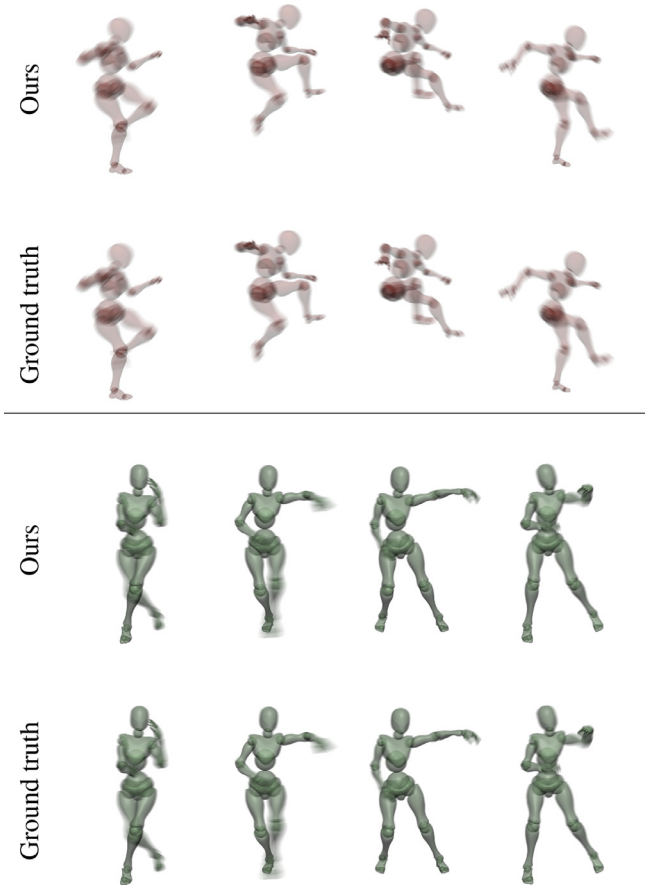

Fig. 12. Quality comparison with ground truth. Transparent characters running (top) and dancing (bottom).

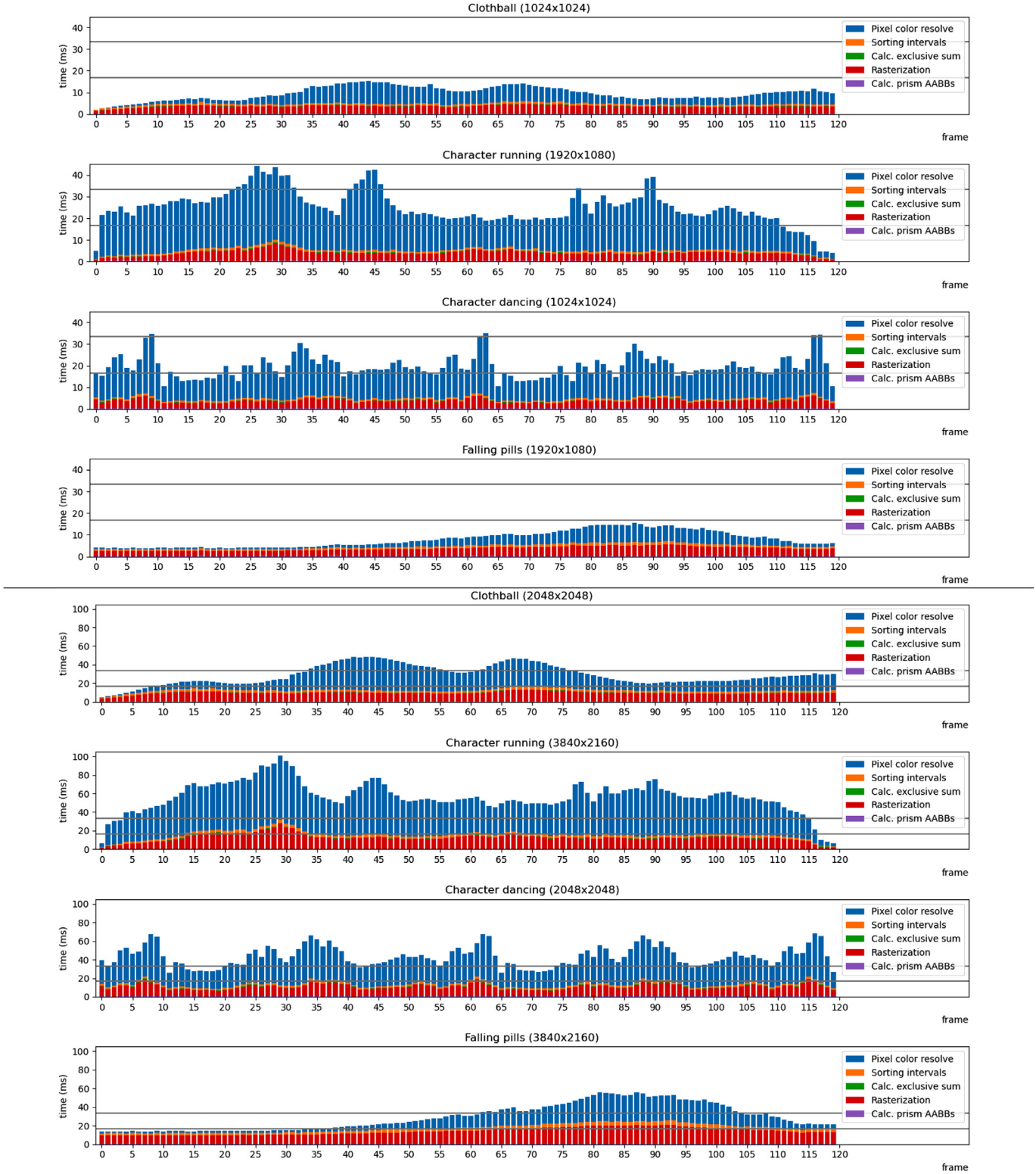

Fig. 13. (Top) Per frame timings for tested scenes. For screen resolutions of 1920×1080 and 1024×1024 (top), the timings generally stay within 30 frames per second and within 60 frames per second for some scenes. For resolutions of 3840×2160 and 2048×2048 (bottom), timings go up to 100 ms in the character running scene. The grey horizontal lines indicate 60 and 30 FPS respectively.

the red-marked lines in Algorithm 1 can be removed. In some pathological cases the image quality loss is significant, such as the one shown in Fig. 11. In our benchmark scenes, the visual difference is mostly undetectable by the human eye, while the time per frame difference is only about 10%.

For quality evaluation, we compare with a brute force, ground truth implementation with 1000 iterations per frame, as well as a fast real-time post-process implementation [1]. The motion blur produced by our method is noise-free and virtually indistinguishable from the ground truth, as shown in Figs. 1, 9, and 12.

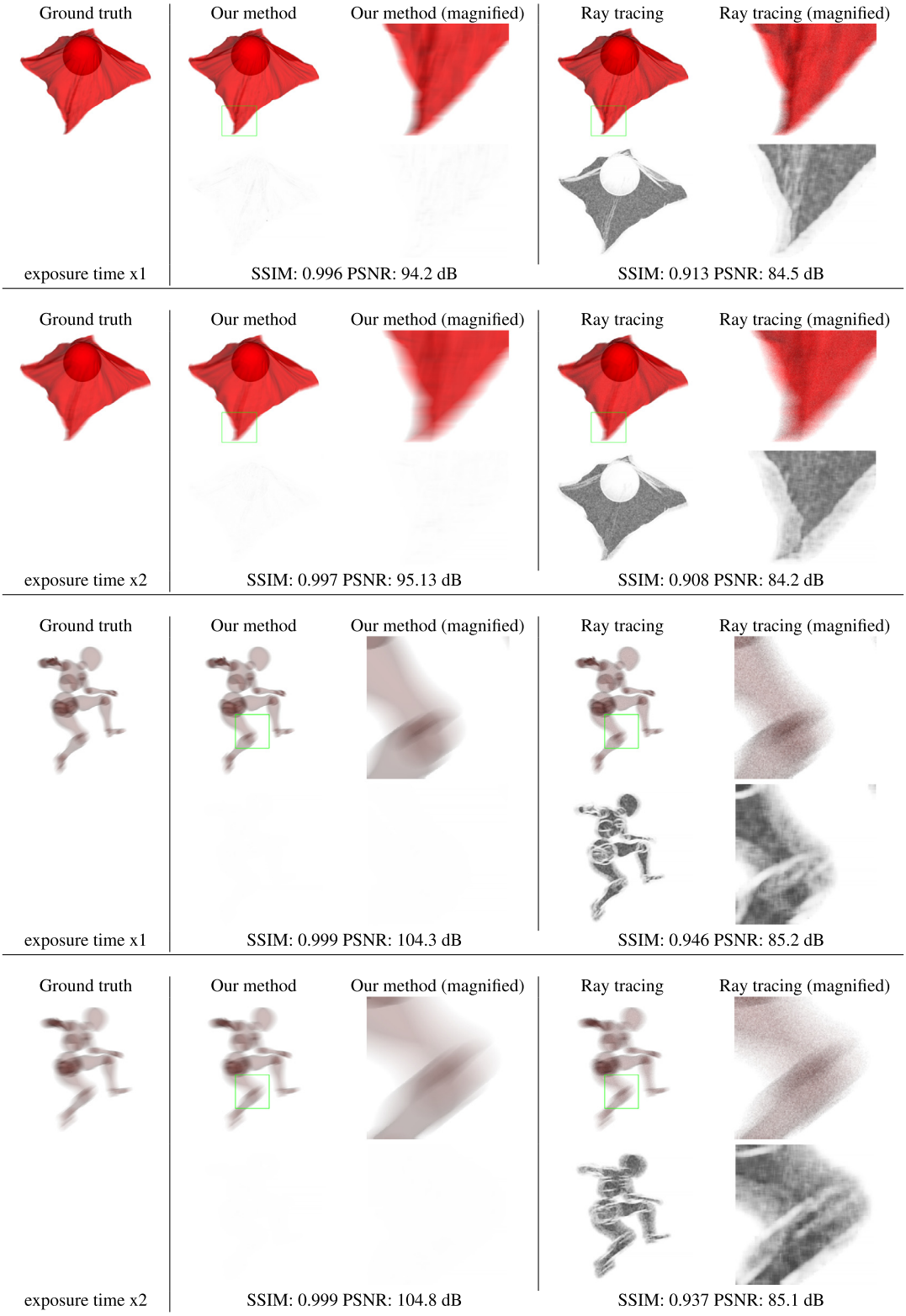

Fig. 14. Qualitative comparison between ground truth (left), our method (middle) and stochastic ray traced motion blur, 128 samples per pixel (right), at different exposure times. Below each image, the normalized difference between the synthesized result and the ground truth is shown as a grayscale image, together with the structural similarity index measure (SSIM), and the peak signal-to-noise ratio (PSNR). Our results are indistinguishable from the ground truth, and are obtained one order of magnitude faster than stochastic ray traced motion blur.

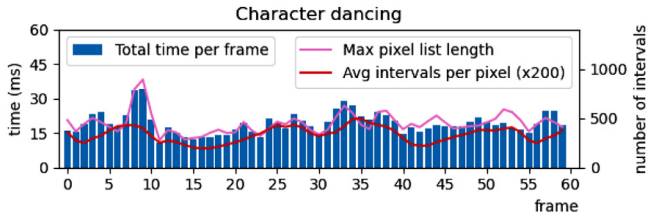

Fig. 15. Time per frame of the first 60 frames of the character dancing scene related with interval counts per frame. The time per frame is largely dependent on the number of intervals per frame and on the length of the longest pixel lists. The average number of intervals considers also empty pixels hence the magnification $\times 200$.

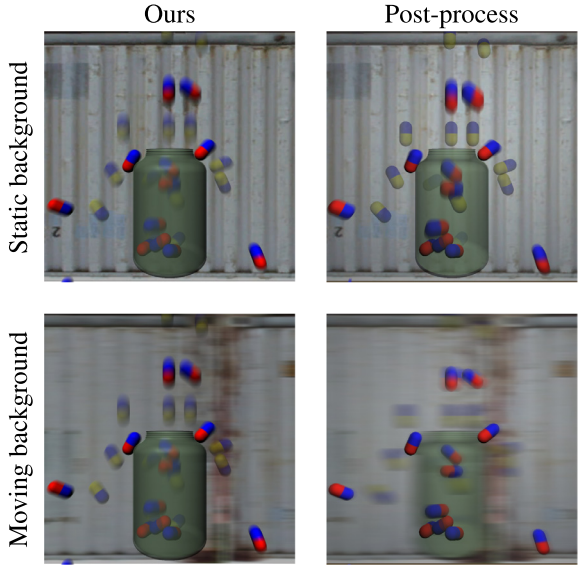

Fig. 16. Quality comparison with a post-process implementation [1]. With a static background (top) the post-process method performs relatively well; only the transparent pills are not blurred. With a moving background, the post-process method produces undesirable blur on the semi-transparent bottle, and the pills are blurred largely in the direction of motion of the moving background instead of their own. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Experimental results are summarized in Table 1 while the frame-by-frame timings of our tested scenes is shown in Fig. 13. Performance scales sub-linearly relatively to the resolution. The relation between frame time and intervals per pixel is outlined in Fig. 15. The pixel color resolve step dominates the time per frame due to it sequentially handling the intervals in each pixel. The main bottleneck is in the pixels with the highest number of intervals.

While the post-process method [1] runs scenes such as *falling pills* in the range of 1ms or less per frame, there are clear cases that it has difficulty in handling. These include cases such as multiple overlapping orthogonal motions and transparency, that are handled well by our method, as shown in Fig. 16. Note that our method supports order-independent transparency, while the post-process method is limited to object order-dependent transparency, without layer information. In the post-processing method, transparent objects do not contribute to the motion vectors used to create blur but are affected by them. The transparent bottle is thus ignored when calculating the motion blur of the opaque pills (red and blue) inside it, but has blur applied to it by motion vectors generated from the moving background, while the transparent pills (yellow and blue) are not blurred in their direction of motion.

Our method is primarily optimized for rendering motion blur with transparent objects. Thus, it is not possible to directly compare with the performance of a method such as Hong and

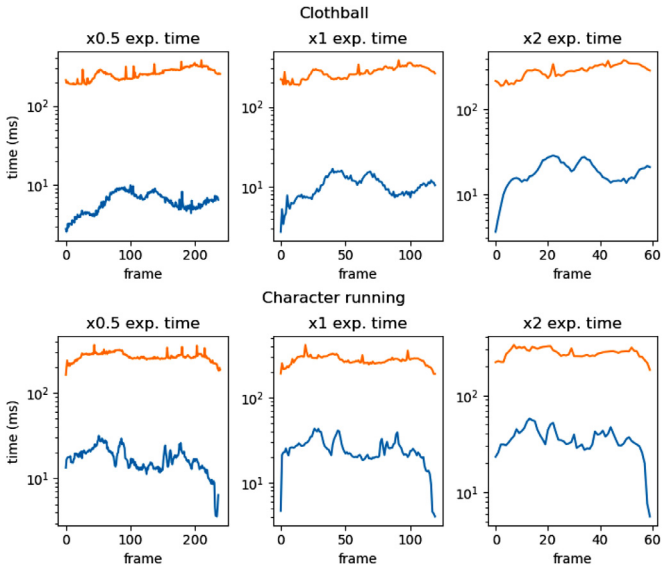

Fig. 17. Per frame timings with varying exposure times of our method (blue) against the per frame timings of ray traced motion blur with 128 samples per pixel (orange) excluding the BVH build time. Our method achieves roughly an order of magnitude faster frame timings. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Oh [3] that takes advantage of the assumption that all objects are opaque. Their paper also does not provide information about depth complexity in their tested scenes, which makes it difficult to argue about relative performance based on tables alone. Their scenes max out at 70 intervals per pixel, which is significantly lower than all the scenes we tested except for one. Given that the performance of these methods depends largely on depth complexity, our method seems to compare favorably.

In order to compare our method with implementations that take advantage of modern hardware's ability to accelerate ray tracing, including BVH traversal and hardware ray/triangle intersection tests, we have implemented stochastic ray traced motion blur in Optix 6.5 [29] and tested on the same hardware. As depicted in Fig. 14 we achieve better image quality than the ray traced method produces with 128 samples per pixel while, as Fig. 17 shows, being roughly an order of magnitude faster, even when excluding the BVH build time. In general, longer exposure time has a negative impact on performance. This correlation, however, seems to be stronger in our method than in the ray traced method.

5.1. Limitations

Our approach has three main limitations. Firstly, our prisms may not always faithfully represent the true trajectory of a rigid triangle, since we use linear motion vectors. An obvious such case include rotations [20]. Secondarily, linear interpolation of texture coordinates between entry and exit points is an approximation. In fact, the *uv* coordinates may follow a curved path in texture space and time. Gribel et al. [2] show that the texture coordinates become rational polynomials of degree two in t. Similar to how Gribel et al. approximates depth using a linear depth function per time partition, our anisotropic texture lookups will approximate the texture-color integration linearly in texture space for the whole interval.

Thirdly, lighting is computed with just the start normal of every interval to conserve memory bandwidth. This approximation and the linear *uv* interpolation introduce an error in the shading, which, however, was not visually noticeable in all our test scenes.

5.2. Discussion

The color-resolve pass (Algorithm 1) can be seen as a deferred-shading step where the final colors are computed in the call to ResolveIntervalRange(). This function currently requires access to all relevant information to compute the final surface shading, in our case: normals, positions, lights, and UVs. We store most of this information within each interval for cache-locality reasons. Further data can be added at a linearly increased cost of the sorting step and overall bandwidth. At some point, indices into a separate buffer may be faster.

A traditional deferred shading pipeline instead typically uses a G-buffer. If transparency is supported, the G-buffer may contain an array per pixel of all the visible semi-transparent fragments.

One way to combine motion-blurred objects with a deferred-shading pipeline for non-motion-blurred objects could be to create an interval, I_j, for each such G-buffer fragment, j, with $z_s = z_e = z$, t_s=0, t_e=1, $uv = j$, *texID=GBuffer*, where z is the depth value for the G-buffer fragment. These intervals are then merged with the intervals from motion blur before the interval-sorting step. If the G-buffer does not support transparency, or if motion-blurred objects are fully in front of or behind the G-buffer pixels, then its colors could be precomputed by a deferred-shader pass and stored with its created intervals.

6. Conclusion

We have presented an efficient GPU rasterization-based method for analytical noise-free motion blur. By representing dynamic triangles as prisms and ray tracing their surfaces in clip space, followed by sorting, and finally color resolving depth-time intervals, our method gives results very similar to the brute force reference while producing superior quality images compared to post-process methods. Our method can deal with cases that are difficult for post-process methods, such as transparency and conflicting motion, and for the scenes we have tested, it runs in real-time, generally higher than 30 frames per second at 1080p.

In future work, we would like to add shadowing and improve performance further by parallelizing the color resolve of each pixel. Back-face culling could be made accurate by implementing the method described by Munkberg and Akenine-Möller [10]. For slightly better precision, a watertight version of the ray/triangle intersection algorithm could be used [30]; unfortunately, to the best of our knowledge there is not yet a watertight ray-bilinear patch intersection algorithm.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Swedish Research Council under Grants 2015-05345 and 2014-4559.

References

[1] Guertin J-P, McGuire M, Nowrouzezahrai D. A fast and stable feature-aware motion blur filter. In: Proceedings of the high performance graphics. Eurographics Association; 2014. p. 51–60.
[2] Gribel CJ, Doggett M, Akenine-Möller T. Analytical motion blur rasterization with compression. In: Proceedings of the high performance graphics. Eurographics Association; 2010. p. 163–72.
[3] Hong M-P, Oh K. Real-time motion blur using extruded triangles. Multimed Tools Appl 2018;77(11):13323–41.
[4] Korein J, Badler N. Temporal anti-aliasing in computer generated animation. SIGGRAPH Comput Graph 1983;17(3):377–88.
[5] Haeberli P, Akeley K. The accumulation buffer: hardware support for high-quality rendering. SIGGRAPH Comput Graph 1990;24(4):309–18.
[6] Akenine-Möller T, Munkberg J, Hasselgren J. Stochastic rasterization using time-continuous triangles. In: Proceedings of the ACM SIGGRAPH/eurographics graphics hardware, GH '07. Eurographics Association; 2007. p. 7–16.
[7] Fatahalian K, Luong E, Boulos S, Akeley K, Mark WR, Hanrahan P. Data-parallel rasterization of micropolygons with defocus and motion blur. In: Proceedings of the high performance graphics. Eurographics Association; 2009. p. 59–68.
[8] Brunhaver JS, Fatahalian K, Hanrahan P. Hardware implementation of micropolygon rasterization with motion and defocus blur. In: Proceedings of the high performance graphics. Eurographics Association; 2010. p. 1–9.
[9] Boulos S, Luong E, Fatahalian K, Moreton H, Hanrahan P. Space-time hierarchical occlusion culling for micropolygon rendering with motion blur. In: Proceedings of the high performance graphics. Eurographics Association; 2010. p. 11–18.
[10] Munkberg J, Akenine-Möller T. Backface culling for motion blur and depth of field. J Graph Tools 2011;15:123–39.
[11] Vaidyanathan K, Toth R, Salvi M, Boulos S, Lefohn A. Adaptive image space shading for motion and defocus blur. In: Proceedings of the high-performance graphics. Eurographics Association; 2012. p. 13–21.
[12] McGuire M, Hennessy P, Bukowski M, Osman B. A reconstruction filter for plausible motion blur. In: Proceedings of the interactive 3D graphics and games. ACM; 2012. p. 135–42.
[13] Guertin J-P, Nowrouzezahrai D. High performance non-linear motion blur. In: Lehtinen J, Nowrouzezahrai D, editors. Proceedings of the symposium on rendering – experimental ideas & implementations. Eurographics Association; 2015.
[14] Nalbach O, Arabadzhiyska E, Mehta D, Seidel H-P, Ritschel T. Deep shading: convolutional neural networks for screen space shading. Comput Graph Forum 2017;36(4):65–78.
[15] Brooks T, Barron JT. Learning to synthesize motion blur. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR); 2019.
[16] Luo X, Salamon NZ, Eisemann E. Adding motion blur to still images. In: Proceedings of the graphics interface; 2018.
[17] Lancelle M, Dogan P, Gross M. Controlling motion blur in synthetic long time exposures. Comput Graph Forum 2019;38(2):393–403.
[18] Luo X, Salamon NZ, Eisemann E. Controllable motion-blur effects in still images. IEEE Trans Vis Comput Graph 2020;26(7):2362–72.
[19] NVIDIA. NVIDIA AMPERE Whitepaper; 2020a. https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf. (accessed November 26, 2020)
[20] Gribel CJ, Munkberg J, Hasselgren J, Akenine-Möller T. Theory and analysis of higher-order motion blur rasterization. In: Proceedings of the high-performance graphics. Eurographics Association; 2013. p. 7–15.
[21] Porter T, Duff T. Compositing digital images. SIGGRAPH Comput Graph 1984;18(3):253–9.
[22] Shkurko K, Yuksel C, Kopta D, Mallett I, Brunvand E. Time interval ray tracing for motion blur. IEEE Trans Vis Comput Graph 2017;PP(99).
[23] Brochu T, Edwards E, Bridson R. Efficient geometrically exact continuous collision detection. ACM Trans Graph 2012;31(4).
[24] Reshetov A. Cool patches: a geometric approach to ray/bilinear patch intersections. Berkeley, CA: Apress; 2019. p. 95–109.
[25] McGuire M, Enderton E, Shirley P, Luebke D. Real-time stochastic rasterization on conventional GPU architectures. In: Proceedings of the high performance graphics. Eurographics Association; 2010. p. 173–82.
[26] Maule M, Comba JLD, Torchelsen R, Bastos R. Memory-efficient order-independent transparency with dynamic fragment buffer. In: Proceedings of the SIBGRAPI graphics, patterns and images; 2012. p. 134–41.
[27] Hou K., Liu W., Wang H., Feng W.. Fast segmented sort on GPUs. 2017, p. 1–10.
[28] NVIDIA. CUB 1.8.0; 2020b. http://nvlabs.github.io/cub/. (accessed November 26, 2020)
[29] NVIDIA. Optix 6.5; 2020c. https://developer.nvidia.com/optix. (accessed November 26, 2020)
[30] Woop S, Benthin C, Wald I. Watertight ray/triangle intersection. J Comput Graph Tech (JCGT) 2013;2(1):65–82.